Ömer Furkan Tercan

Exploring Software Refactoring Decisions in a Lean Startup

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 30.11.2015

Thesis supervisor:

Assoc. Prof. Casper Lassenius

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Ömer Furkan Tercan

Title: Exploring Software Refactoring Decisions in a Lean Startup

Date: 30.11.2015 Language: English Number of pages: 8+54

Department of Computer Science and Engineering

Professorship: Software Engineering Code: T-76

Supervisor and advisor: Assoc. Prof. Casper Lassenius

Software systems are continuously forced to evolve as they can not resist change. Quality typically degenerates as a software is subjected to change during the course of its lifetime. In this process, software quality must be audited, secured, and maintained, whereas maintaining such a system demands continuous refactoring.

Researchers have contributed widely in the area of software refacting. Fowler and Beck have introduced 22 problematic code smells considered as drivers for refactoring decisions, whereas Brown et al. have identified development anti-patterns known to make systems harder to maintain.

In spite of the wide contribution in the field, there is still little evidence to justify the usage of refactoring drivers. This study aims to contribute in this research gap by finding evidence on how software practitioners behave when making refactoring decisions.

To achieve its objective, this study initially conducts a literature review on the drivers for refactoring decisions, i.e., code smells and anti-patterns. Further, it examines relevant literature exploring the usage of these drivers.

Finally, we conduct a case study introducing new empirical evidence on how software practitioners make use of refactoring drivers. We further discuss the relation between our empirical findings and the examined literature.

Our key findings indicate that the code smells and anti-patterns found in the literature are not enough to be used as a basis for refactoring decisions. Drivers related to code documentation and style have been mostly neglected in the literature, whereas together they were the underlying reason for the 45% of all refactoring decision made in the case company.

Keywords: software refactoring, code smells, anti-patterns, quality attributes

Preface

This study was conducted while I work in a lean startup company to help improve their agile development process. I would like to thank all my colleagues for their kind support and valuable input when making this case study.

I would like to thank my supervisor Casper Lassenius for his guidance, encouragements and suggestions. His mentoring was very valuable in finishing this work.

Additionally, many thanks to all friends for just hanging in there.

Finally, I would like to express my deepest gratitude for being together with my dear wife, wonderful children and big-hearted parents.

Otaniemi, 30.11.2015 Ömer Furkan Tercan

Contents

Glossary

IDE Integrated development environment. 8, 37, 39

LOC Lines of code. vii, 29, 30

RCS Revision control system. 31

List of Tables

List of Figures

1 Introduction

In this chapter, section 1.1 presents the study background and motivation. The following section 1.2 introduces the objective and research questions. Finally, section 1.3 summarizes the structure and scope of this document.

1.1 Background and motivation

Software systems are continuously forced to evolve due to ongoing system requirements. During this process, software quality must be audited, secured, and maintained. Otherwise, quality typically degenerates as software systems are constantly subjected to change.

Lehman's laws of software evolution [1] state that, a functionality increment of a system always brings a corresponding decrease in the quality and an increase in the internal complexity. Therefore, in order to maintain software quality, each and every functionality increment of a system should be accompanied by a number of supplemental activities. Such maintenance activities imply continuous improvement on software code base, namely refactoring. For instance, in order to maintain and re-develop its products, Microsoft reserves 20% of its development effort for software refactoring [5].

Software refactoring is a process of change and improvement. It changes and improves the internal structure of a software system without altering its external behavior [6]. This activity improves code understandability, allows easier debugging and accelerates software development. Refactoring improves software quality. Typically, it achieves this by resolving quality defects such as code smells, development anti-patterns and other anomalies [21].

Refactoring naturally fits within the process of software reengineering [10], which aims to restructure legacy software. Therefore, refactoring as a concept was previously practiced as part of software restructuring. When object-oriented design emerged as a contemporary concept in the early nineties, the term refactoring became more prominent [6].

Researchers have contributed with a wide range of studies in the field of software refactoring. Fowler and Beck [6] have introduced 22 problematic code structures within the object-oriented context that points to refactoring needs. Mäntylä et al. [11] have outlined a classification for these 22 smells and further investigated the correlation among them by conducting an empirical study. Similarly, Brown et al. have listed 14 development anti-patterns in his book[4] and discusses design flaws that make systems harder to maintain.

Researchers have also worked with software practitioners and have conducted surveys

and empirical studies to examine refactoring decisions. Siy and Votta [9] have studied data of 130 code inspection sessions and have introduced four groups of code maintenance drivers, namely documentation, style, portability, and safety. They have also presented empirical evidence on how software practitioners perceive these drivers. In their survey [23], Yamashita and Moonen have inspected code smells from the practitioner's perspective and have presented a prioritized list of smells. In addition to these, there are empirical studies that introduce automated tool support for detecting code smells [7, 8, 13].

Even though there is significant amount of related work identifying software refactoring drivers, there is still little evidence to justify the usage of these drivers. In particular, the study of the human perception of what a code smell is and how to deal with it has been mostly neglected in the past. In order to understand the usage of these drivers, the software practitioner's perception of refactoring drivers can be investigated. Having such focus on the human factor, further research on the drivers behind refactoring decisions can bring significant contribution to this research gap.

Motivated by the prior studies and the identified research gap, this thesis investigates the performed refactoring decisions in a lean startup software company. This is achieved through a case study. The results of the case study is analyzed, discussed and the most significant findings are compared with the existing scientific knowledge on refactoring decisions. This study also discusses relevant issues which can be investigated further in the future.

This study contains an overview of software refactoring in Chapter 2. This chapter presents refactoring drivers found in literature, namely code smells and development anti-patterns. It introduces a study of related surveys and empirical work which investigates refactoring decisions and practitioner perceptions.

1.2 Objective and research questions

According to the motivation stated in section 1.1, there is a need to investigate and reveal the correlation between how literature have conceptualized drivers for software refactoring decisions and how it is actually perceived by software practitioners. Accordingly, an empirical study can be conducted investigating refactoring decisions in a case company. The empirical findings can then be analyzed and compared with the scientific knowledge regarding software refactoring decision.

This motivation leads to the main study objective:

Objective

Study software refactoring decisions, particularly how it is conceptualized in the literature and comparably how it is perceived by software practitioners.

The following research questions were derived based on the main objective.

Research questions

RQ1: What does the literature say on how refactoring decision are made?

This research question attempts to investigate relevant literature to form a basis on refactoring decision making, In particular, it aims to identify the underlying drivers for refactoring decisions. This research question can be further investigated in two sub-questions:

- **RQ1.1:** What does the literature say about the terminology used in identifying drivers for refactoring decisions?
- **RQ1.2:** What quantitative findings does the literature reveal regarding these drivers?

RQ2: How does developers make refactoring decisions?

This research question is of exploratory nature and it aims to find new empirical evidence on the drivers for refactoring decisions. It takes into account that all software development activities are highly human dependent. Therefore, the developer's perception is kept in the research foreground. This research question can be divided into two sub-questions:

- **RQ2.1:** What are the identified refactoring drivers from the viewpoint of software developers?
- **RQ2.2:** How does their viewpoints relate to the literature?

1.3 Structure and scope of the thesis

Chapter 1 presents the background and motivation of the thesis. This is accompanied by the main objective and research questions.

Following the introduction, Chapter 2 introduces an overview on software refactoring. It presents the refactoring drivers found in literature. This chapter concludes by

outlining the most relevant surveys and empirical studies that are investigating refactoring drivers and software practitioner perceptions.

Chapter 3 elaborates the main study objective and research questions. Furthermore, it presents the case study environment and the case study methodology.

Chapter 4 reports the case study results and discusses the findings. Collected and analyzed empirical data on refactoring decisions are presented. This is accompanied by a discussion on the comparison of the literature findings and case study findings.

Finally, Chapter 5 concludes the outcome of the thesis. It presents the potential validity threats and indicates future research areas.

2 Software refactoring

The term software refactoring was introduced for the first time by William F. Opdyke in his Ph.D. dissertation [3]. It became more practical and commonly used after the publication of the book Refactoring: Improve the Design of Existing Code, which was written by Martin Fowler in the year 1999 [6].

Refactoring is the process of making code level changes to improve its internal structure. The cumulative effect of these changes can radically improve software design [6]. Software refactoring is a form of code modification, used to improve the software structure in support of subsequent extension and long-term maintenance, in most cases having the goal to transform code without impacting correctness [4]. The basic principle in refactoring process is reorganizing classes, variables and methods across the class hierarchy to facilitate future adaptations and extensions, so that the source code can have better structure, readability and understandability [24].

Refactoring is important in order to maintain and improve code evolvability [15]. Particularly in agile development, refactoring is considered as a key practice and applied as a building block in order to compensate the lack of upfront design [15].

Refactoring is also performed to improve software readability. It improves code readability similarly to other quality attributes without introducing any observable behaviour change to the software. Any user, whether an end user or another developer, can not tell that things have changed [6].

Refactoring can improve code maintainability. It can assist the code in conforming to coding standards, help minimize redundancies, improve software security, and enable system portability [9].

An outline of the most significant benefits of software refactoring is presented in Table 1.

According to a survey on software refactoring [14], the refactoring process consists of a number of sequential activities presented in Table 2.

This thesis mainly discusses the first two activities; *identifying the areas for refactoring* (drivers) and *determining which refactoring solutions should be applied*. Accordingly, this section presents refactoring drivers addressed in the literature, namely code smells and development anti-patters. The Proposed refactoring solutions are also described under each refactoring driver.

Table 1: Benefits of software refactoring

Benefit	**Explanation**
Removes duplicated code	Programs having duplicated code are difficult to modify, as they need be modified in multiple ways for a single change
Improves the design	The software design will decays unless developers perform regular refactoring as they make code changes to realise short term goals
Makes the code easier to understand	Even if codes are written to instruct computers, there is always someone who will try to read. Even the code owner might have problems understanding the logic after several months if code understandability is neglected.
Helps to program faster	A good design is important to maintain speed in software development. Refactoring improves software quality, thus reduces commonly faced issues such as extendability and maintainability.

Table 2: Refactoring activities

	Activity
1	Identify the areas for refactoring
2	Determine which refactoring solutions should be applied to the identified areas
3	Guarantee that the applied refactoring preserves behavior
4	Apply the refactoring
5	Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort)
6	Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests, etc.)

2.1 Drivers for refactoring decisions

Identifying where to refactor in a software system can be difficult. Particularly, locating problematic parts known as code smells and anti-patterns can be quite

challenging. However once detected, these problematic parts can be used as drivers for refactoring decisions.

In this chapter we identify the mainly referred drivers for refactoring decisions in the literature. In Chapter 4, we will then compare and discuss these findings together with our empirical results.Drivers for refactoring decisions are outlined in two categories; code smells [6] and development level anti-patterns [4].

2.1.1 Code smells

Code smells or *bad smells* are structures in code that suggest the possibility for software refactoring [6]. A Code smell does not indicate how to solve a problematic peace of code, but it narrows down the number of potential refactoring solutions that can be performed. Fowler provides a list of code smells and recommends possible refactoring solutions for each smell. In this manner, determining the drivers (code smells) and solutions would assist developers in making more intelligent refactoring decisions [6]

Fowler identifies code smells that primarily occur in object-oriented design. He recommends refactoring solutions based on object-oriented design patterns for smells occurring in cases of; interacting objects, information hiding, inheritance, interfaces, and polymorphism. Therefore, code smells can be derived based on poorly used, over-used, or misused object-oriented design principles.

As a supporting work, Mäntylä et al. classifies Fowler's 22 smells to make the smells more understandable and to disclose the relationships among them [11]. According to their classification, code smells can be specified in seven categories; bloaters, object-orientation abusers, change preventers, dispensables, encapsulators, couplers, and others. Next, we introduce the smells using this classification.

Bloaters

Bloaters represent something in the code that has grown so large that it cannot be effectively handled [11]. This category include the code smells; long method, large class, primitive obsession, long parameter list, and data clumps.

Long method — the software that lives best and longest are those with short methods [6]. The longer a method body is, the more difficult it is to understand. It is usually more vulnerable to possible bugs due to having more corner cases to cover.

Formerly, programming languages and development environments have deterred people from writing small methods. Programming languages were introducing overhead in subroutine calls and development environments were unable to easily

switch context between routines and subroutines. However, modern languages and integrated development environments (IDE), have overcome these shortages. The observable effect is that developers can be more comfortably use decomposed methods to improve understandability, robustness and modularity.

Several solutions can be proposed to refactor long methods [6]:

- Replace comments with methods to reduce the semantic distance between what a method does and how it does it.
- Shorten methods by extracting code parts that seems to go nicely together into their own subroutines.
- To overcome method extraction issues when when there are lots of temp variables, conditionals and loops, decompose them into utility methods
- To overcome method extraction issues when a method has a long parameter list, introduce parameter objects.
- If a method still can not be decomposed easily, turn the method into its own object so that all the local variables become fields of it's own and then decompose the method into other methods on the same object.

Large class — reveals to much responsibility. When a class is trying to do too much, it often shows up too many instance variables. This brings a lot of duplicated code into existence. A large class is also commonly exposed to changes for different reasons. This introduces extra complexity when software changes are required.

Two refactoring solutions are suggested for a class with too much code – a prime breeding ground for duplicated code, chaos, and death [6].

- Extract class and split the responsibility.
- Extract subclass for a subset of features. If a class has features that are used only in some instances, extract that subset into a subclass.

Primitive obsession — most programming languages have two data types – primitives and objects. Primitive type is a simple type holding a single value to represent a single data item. Objects on the other hand, allows to structure data items into meaningful groups. If a data item needs to be represented rather comprehensively, by using primitives one needs to extend the owning class with methods to satisfy the requirements. This can quickly result in code smells of duplication and feature envy.

In order to represent comprehensive data items without introducing code smells several refactoring solutions have introduced [6]:

- Replace data value with object. Turn the data item into an object once it requires additional data or behaviour.
- If the data value is a type code and does not affect class behaviour, replace it with class. By wrapping numeric type codes and enumerations into a class and providing static factory methods for handling them, readability of the code can be improved and potential bugs can be prevented.
- If the type code is affecting class behaviour, replace type code with subclass. Form an inheritance structure having a subclass for each type code. Following this, replace conditionals that depend on the type codes with polymorphism

Long parameter list — having method parameters is an alternative to having global variables. However, this is only true in our early programming days. However in an object-oriented context, instead of using a long parameter list or declaring global variables, a method can request data from other objects. Thus, instead of passing everything a method needs, it is enough to pass parameters so that the method can retrieve relevant data.

Refactoring solutions are introduced for long parameter lists [6] that reduces the likelihood of having methods that are inconsistent, difficult to use, understand, maintain and change:

- Replace parameter with method. If a method can get a value that is passed in as parameter by another means, it should [6]. If a method parameter is formed of an expression, the parameter can be removed by extracting the expression intro a subroutine and calling it directly from the method.
- Use preserve whole object. If several values of an object is passed as parameters into a method, instead, the whole object can be passed as a parameter or an appropriate method of the whole object can be invoked that returns required values.
- Introduce parameter objects based on group of data that naturally go together.Data clumps which are typically passed together as parameters, can be replaced with objects that wraps them.

Data clumps — are one of the causes of long parameter lists, long methods, and large classes. Until they are extracted into a class, data items that naturally go together float within code and cause duplication, inconsistency, maintainability and readability issues.

This code smell can be resolved by using class extraction to split responsibility, previously explained on *large class* code smell. Data clumps can be further served as parameter objects or, preserved as whole objects. Both refactoring solution was elaborated on *long parameter list* code smell.

Object-orientation abusers

This category of smells is related to cases where the solution does not fully exploit the possibilities of object-oriented design [11]. Switch statements, temporary field, refused bequest, alternative classes with different interfaces, and parallel inheritance hierarchies are considered in the object-orientation abusers category.

Switch statements — result in duplicated code and a significant maintainability issue when code is subject to change. It causes duplication since the same statement typically exists in multiple places. In conjunction with duplication, it significantly prevents change. If a switch statement needs an additional clause, the change needs to be applied in all places where the statement is used.

The proposed refactoring solutions [6] indicate using the object-oriented notion of polymorphism.

- If a switch statement occur on a type code, similar to the refactoring solutions proposed on the *primitive obsession* code smell, an inheritance structure can be formed having a subclass for each type code. Following this, switch statements can be replaced with polymorphism.
- If there are only few cases which makes using polymorphism an overkill, add an explicit method for each conditional case in the switch statement.

Temporary field — smell occurs in the case where a class variable is only set and used in certain cases. For instance, a temporary variable could be defined in the class scope, whereas it should only be defined a method scope. This type of smell make it difficult to understand the duty of the temporary variable.

In order to resolve this type of code smell, temporary fields and the methods that require them can be extracted into a new class [6]. The new method object can then be used by the original class.

Refused bequest — smell is a sign of an inappropriate inheritance hierarchy. Subclasses get to inherit the methods and data of their parents [6]. This smell occurs when there is clash between what subclass expects to reuse and what it inherits from the parent class.

There are two refactoring solutions proposed to resolve the conflict on the inheritance hierarchy:

- Heading towards having an abstract superclass by pushing down fields and methods to subclasses.

- Removing the inheritance hierarchy and replacing inheritance with delegation. This solution can be used in the case where the subclass is refusing the superclass interface and causing the inheritance hierarchy to be useless. Instead, the superclass can be declared as a class variable and used appropriately.

Alternative classes with different interfaces — smell is a sign of not using inheritance properly on closely related classes. These classes contain similar features which can be considered as duplicate code and a preventer for change.

As a refactoring solution the commonalities can be extracted into a superclass. Consequently, the alternative classes can be incorporated in a inheritance hierarchy.

Parallel inheritance hierarchies — smell needs to be fixed by using a proper inheritance hierarchy. Tightly coupled classes result in tightly coupled inheritance hierarchies. This results as a special case of the *shotgun surgery* code smell having duplicated code[6].

The proposed refactoring solution to eliminate duplication is to make sure that instances of one hierarchy refer to instances of the other. This means decoupling classes that have a parallel inheritance hierarchy. Achieving this requires moving methods and fields from one class to the other and using only references. As a result the inheritance hierarchy on the referring class will disappear [6].

Change preventers

Change preventer code smells imply code structures that considerably hinder the modification if the software. [11]. Ideally according to [6], there is a one-to-one link between common changes and classes. This category contains divergent change and shotgun surgery code smells:

Divergent change — is one type of smell that causes difficult to modify code. It occurs when a class acts feature envy containing various responsibility, as a result being exposed to change in different ways for different reasons. Any change to handle a variation should change a single class, and all the typing in the new class should express the variation [6].

As a refactoring solution, one must identify all changing blocks for a particular cause and extract them into a new class.

Shotgun surgery — in the other code smell type that causes difficult to modify code. It occurs when a change has an effect on multiple locations. It is considerably difficult to modify certain part of a software, if it requires multiple changes in multiple classes.

Proposed refactoring solution suggests moving methods/fields to another class and

access them via reference. Thus, code blocks affected by a particular change is put into a single class.

Dispensables

This type of code smells indicate existing unnecessary peaces in the code. This could be related to having lazy, redundant, or dead peaces in the code. Interestingly, Fowler [6] do not present a smell for dead code. Dead code is code that is left behind due to legacy reasons, evolved software, changed requirements, or refactoring. Dead code hinders code comprehension and makes current code structure less obvious [11].

Dispensable code smells include, lazy class, data class, duplicated code, and speculative generality.

Lazy class — is a class that is not doing enough work and suitable to be eliminated with a short effort.

There are two refactoring solutions proposed to eliminate a lazy class:

- Code in a nearly useless class can be moved to another class.
- In a inheritance hierarchy, if a subclass is not adding any value, the hierarchy can be collapsed. The methods and fields in the subclass can be pulled up to the parent class.

Data class — is a simple class that only have data fields and getter/setter methods for the fields. Similar to a lazy class, it does not have enough responsibility. Yet in this case, it is more suitable to embrace extended functionality.

There are several building blocks to extend the responsibility of a data class:

- First of all, encapsulation should be applied to its publicly available variables.
- Setter methods should be removed for variables that should not be modified.
- The behaviour of the outsider methods that invoke the getter and setter methods of the data class can be moved to the data class.
- Finally, encapsulation can be applied to its methods that are not used by other classes.

Duplicated code — smell indicates having the same code structure in multiple places. When a modification is required in such a code structure, the same change need to be applied in multiple locations.

The proposed refactoring solution recommends unifying recurring code in a method. This can be achieved in steps;

1. Extract duplicated code as a separate method.
2. Decide where the new method should belong.
3. Ensure that only this methods is called from all locations that seek the functionality.
4. Remove its duplicates.

Speculative generality — smell occurs when unnecessary ability is introduced to the code for the sake of speculative reasoning. This often results in a code with unused parts that is difficult to understand and maintain.

The refactoring solutions used to eliminate speculative generality include:

- If abstract classes are used that does not add any value, inheritance hierarchy can be collapsed.
- Unnecessary delegation can be resolved by moving the functionality indoors.
- Unused method parameters should be removed.
- Methods having difficult to understand abstract names should be renamed.
- Identify methods/classes that are only used in test cases. Remove them in case they are not supportive for test cases that exercise legitimate functionality.

Encapsulators

Encapsulation means hiding internal details from other classes. [6]. Code smells in this category reveal issues on the way objects, data, or operations are accessed. The smells in this category are message chains and middle man.

Message chain — smell usually indicates a coupling problem. When there is a chain of tightly coupled classes, modifying one requires changes on others.

Refactoring solution proposes hiding delegates using the following steps:

- Add a simple delegate method on the middle man.
- Adjust the caller class to call the delegate method on the middle man.

- Remove direct access to the delegate object on the middle man.

.

Middle man — is a result of applying excessive amount of encapsulation and delegation. In this case, a class can become a middle man.

Different refactoring solutions can be applied in this case, depending on the condition of the class:

- If the class is nothing but a middle man, it can be removed, and the caller classes may start accessing its delegates directly
- If only a few method is used as delegates, move those methods to the caller class.
- If you have many simple delegations for the entire interface and if there is additional behaviour that stops your from removing the middle man, replace delegation with inheritance and make the delegating class (middle man) a subclass of the delegate. This will allow to extend delegate behaviour without any chaining.

Couplers

High coupling is against object-oriented design principles. It dramatically reduces the ability to modify and reuse software components. Code smells in this category are highly related to object-orientation abuser smells. This category includes, feature envy and inappropriate intimacy smells.

Feature envy — smell means a case where one method have a lot of responsibility and more interested in other classes than the owning class. This smell clearly indicates that such a method belongs elsewhere.

Depending on the method condition, several refactoring solutions can be applied:

- If it is obvious that the method belongs to a particular class, it can be moved. The method should go together with the data that it uses and the idea is to put things together that change together. This rule of thumb can be used in deciding a correct location for such a method.
- In case only one part of the method suffers from envy, that part can be extracted and moved as a new method to the relevant class.

Inappropriate intimacy — means that two classes are tightly coupled with each other [11]. In such a case, following refactoring solutions can serve several approaches:

- Moving over-intimately used methods or fields to the correct class and improve decoupling.
- Change bidirectional associations to unidirectional. Bidirectional associations introduce added complexity of maintaining the two-way links and ensuring that objects are properly created and removed [6]. Additionally, they are usually not perceived as natural by developers, so they often are a source of errors [6].
- Hide delegate to reduce intimacy and introduce a middle man class for accessing delegates.
- If inheritance is causing inappropriate intimacy, replace it with delegation. At the subclass, create a field for the parent class, adjust methods to delegate to the parent class, and remove the inheritance hierarchy.

Others

This category contains code smells that do not fit into any of the above categories. It covers incomplete library class and comments.

Incomplete library class — code smell occurs in case a library is in a bad form and impossible to modify. Under this circumstances, as a refactoring solution, one can introduce foreign methods or classes as a local extension. This way, they could be further refactored and modified according to needs.

Comments — in code can be an indication of a bad smell nearby. The best practice is to go through code comments, identify the bad smells nearby and refactor. Refactoring nearby code smell can transform code comments to be unnecessary.

Possible refactoring solutions include:

- If a block of code requires comment, extract that block into a new method and assign a descriptive name.
- If a method body is hard to understand and requires comment, rename the method to be more explanatory.
- if a block of code relies on an assumption, make that assumption explicit by introducing an assertion.

On the other hand, comments can be used as a refactoring solution themselves. It can be useful to add comments to describe why something was implemented in such a way, or how it can be improved.

Martin extended the work of Beck and Fowler with an elaboration on a set of design principles and new smells that were advocated by the Agile community [12].

2.1.2 Development anti-patterns

An anti-pattern is defined by Brown et al. [4] as a commonly occurring solution that will always generate negative consequences when it is applied to a recurring problem. In their book, Brown et al. describes anti-patterns in three levels including, development, architecture, and project management. This section only describes development level anti-patterns, since we are only interested in code level refactoring decisions,

Development anti-patters are conjectured in the literature to make systems harder to maintain. Moreover, code smells presented in the previous section can be seen as being symptoms of these anti-patterns. Even though they overlap, there are anti-patterns which are not referred in code smells.

The blob

The blob is found in designs where on class controls all the processing while others act like simple data classes. A blob is an outcome of a procedural design even though it may be represented using object notations and implemented in object-oriented languages. Therefore, as in procedural design, the blob contains the majority of processes while the other objects contain only data.

Some code smells can be symptoms of the blob anti-pattern. The *large class* smell may contain the blob anti-pattern if it controls all the processes. This will leave the rest of the classes to act as a *Data class* or *lazy class*. This anti-pattern can also be a sign of a *Large class* having a lot of *Feature envy* methods, while other *Data classes* having *Inappropriate intimacy*.

Divergent change code smell is another symptom of this anti-pattern. A blob class controlling most of the processes, is exposed most to different changes for different reasons.

The solution includes refactoring the design to distribute responsibilities more uniformly and isolating the effect of changes. Detailed refactoring techniques can be found under related code smells.

Table 3: The blob anti-pattern view

Related smell groups	Bloaters, Change preventers, Dispensables, Couplers
Related code smells	Large class, Divergent change, Data class, Lazy class, Inappropriate intimacy
Refactoring solution	Applying proper decomposition and distributing responsibilities more uniformly to other classes

Lava flow

Lava flow anti-pattern imply dead code and forgotten design information left as legacy in an evolving software. As a consequence, code that are redundant, commented out, or not understood by anyone is left behind in the software, Unless these symptoms are cleaned, the software is difficult to extend, maintain and understand.

Lava flow is clearly related to the *Dispensables* code smell group. However, Fowler [6] does not mention dead code as a code smell.

The refactored solution includes a configuration management process that eliminates dead code and refactors software design toward increasing quality. This configuration management process can be strictly executed as part of the architectural configuration management, however, in agile development process it can be informal and tight to intensive face-to-face communication, pair reviews, and shared code ownership.

Table 4: Lava flow anti-pattern view

Related smell group	Dispensables
Refactoring solutions	In architecture-centric development processes, introducing a configuration management process to eliminate dead code and refactor software design in architecture-centric development processes. In agile development processes, relying on intensive face-to-face communication, pair reviews, and shared code ownership to eliminate redundancies

Functional decomposition

Functional decomposition is a direct match with the *Object-orientation abusers* code smell group. It refers to the misuse of object-orientation and resulting with code that resembles a structural language in class structure.

Table 5: Functional decomposition anti-pattern view

Related smell group	Object-orientation abusers
Refactoring solutions	Reengineering object-orientation

Poltergeists

Poltergeists are classes with very limited roles and effective life cycles. This anti-pattern, depending on its condition, is a direct match to the *Data class* or *Lazy class* code smell.

Table 6: Poltergeists anti-pattern view

Related smell groups	Dispensables
Related code smells	Data class and Lazy class
Refactoring solutions	Reallocating responsibilities to longer-lived objects that help mature data classes or eliminate lazy classes, as a result remove poltergeists

Golden hammer

This anti-pattern appears as one particular programming solution is applied in a diverse range of problems, even though there might be better solutions. The reason could be taking less risk, or neglecting to learn new approaches. As a result, this anti-pattern may introduce inferior performance, scalability, and so on when compared to other available solutions.

None of Fowler's code smells can be seen as a symptom of this anti-pattern.

Table 7: Golden hammer anti-pattern view

Refactoring solutions	There are no direct code level refactoring solutions, however, process level improvements include, managing knowledge transfer, taking more risk and responsibility, and developing competences

Spaghetti code

Spaghetti code appears in a software which contains very less software structure. This anti-pattern is the most widely known design problem and, even though nonobject-

oriented languages are more susceptible, it is also seen in the object-oriented context.

Spaghetti code may exist where there is a *Large class*, a *Long method*, or a *Divergent change* code smell. It may also intro tight coupling, therefore, *Couplers*, namely *Feature envy* and *Inappropriate intimacy* smells may easily exist. Additionally, most of class methods may be utilizing class or global variables instead of having parameters, which will cause a lot of *Temporary field* smells.

The examples can be extended, however, in summary, spaghetti code can expose the software to several group of code smells including, *Object-orientation abusers, Change preventers* and *couplers*.

Table 8: Spaghetti code anti-pattern view

Related smell groups	Object-orientation abusers, Change preventers, Couplers
Related code smells	Large class, Long method, Divergent change, Feature envy, Inappropriate intimacy, and Temporary field
Refactoring solutions	This anti-pattern covers quite a lot of code smells and it is difficult to point specific refactoring suggestions. However, to discard spaghetti code, continuous refactoring and code cleanup should be practiced

Cut and paste programming

Cut and paste programming is a clear blocker of software reuse and change. This anti-pattern is identified by the presence of duplicate segments of code spread through the software. Even though code duplication has a positive effect when to quickly make short-term modifications, in the long term it causes significant, reuseability and maintainability problems.

Cut and paste programming clearly matches with the *Duplicated code* code smell, as well as it is highly related to the change preventer code smell *Shotgun surgery*.

Back-box reuse enables reusing an external class or method without modifying it and without needing to know its implementation. It is an alternative to *white-box reuse*, where developers take a class or method from elsewhere and modify it (i.e., via extending) according to specific needs.

Table 9: Cut and paste programming anti-pattern view

Related smell groups	Change preventers and Dispensables
Related code smells	Duplicated code and Shotgun surgery
Refactoring solutions	Unifying duplicated code, ideally taking advantage of black-box reuse

Other anti-patterns

In addition to the described development anti-patterns, Brown [4] explains several other "mini" anti-patterns including, *Continuous obsolescence, Ambiguous viewpoint, Boat anchor, Dead end, Input Kludge, Walking through a minefield*, and *Mushroom management.*

- *Continuous obsolescence* anti-pattern is somewhat related to the *Incomplete library class* code smell in which developers experience integration issues with external dependencies.
- *Ambiguous viewpoint* anti-pattern is related to the misuse of architectural patterns such as Model-View-Controller(MVC) and object-orientation.
- A *Boat anchor* is a direct match to the *Speculative generality* code smell, which is a peace of software or hardware that serves no useful purpose.
- The *Dead end* anti-pattern appears when there is an external component used within the software and that is no longer maintained and supported by the supplier. In this "dead end" scenario, similar steps can be taken as in the *Incomplete library class* code smell.
- *Input kludge* anti-pattern appears when ad hoc algorithms are employed for handling program's user input, in which case indicates the lack of precaution for problems that may occur due to program input.
- *Walking through a minefield* anti-pattern indicates that, just because a software is released, it does not mean it is production ready. All software products contain defects and unless they are discovered through solid test coverage, bugs can appear on production and cause catastrophic problems.
- *Mushroom management* anti-pattern may appear due to the isolation between developers and end users. If short feedback cycles are not performed, end user expectations can be easily misunderstood.

2.1.3 Summary

Code level refactoring decisions can be made based on code smells. Code smells are design flaws in object-oriented designs that may lead to maintainability issues in the further evolution of the software system **??**. These smells particularly address problems to fulfill object-oriented design principles. In the object-oriented context, Fowler describes 22 code smells together with a set of recommended refactoring solutions.

Table 10 outlines the code smell classification of Mäntylä et al. [11]. This classification assists in fully understanding the code smells and how they relate to each other.

Table 10: A taxonomy of code smells

Taxonomy class	**Code smells**
Bloaters	Long method, Large class, Primitive obsession, Long parameter list, and Data clumps
Object-orientation abusers	Switch statements, Temporary field, Refused bequest, Alternative classes with different interfaces, and Parallel inheritance hierarchies
Change preventers	Divergent change and Shotgun surgery
Dispensables	Lazy class, Data class, Duplicated code, and Speculative generality.
Encapsulators	Message chains and Middle man
Couplers	Feature envy and Inappropriate intimacy
Others	Comments and Incomplete library class

On the other hand, development anti-patterns tend to reveal problems in multiple levels including, code, architecture and development process. Anti-patterns overlap and extend code smells in many occasions. A considerable amount of code smells can be considered as symptoms of anti-patterns. Table 11 presents the relation between anti-patterns and code smells.

Some of the anti-patterns presented in Table 11 do not have any code smell symptoms. This is due to their architectural and process level dependencies. Refactoring such anti-patterns require architectural and process level changes in areas such as knowledge transfer, configuration management, requirements engineering, requirements change management, continuous integration, and continues deployment.

Table 11: Development Anti-patterns and their symptoms (code smells)

Anti-pattern	**Symptoms**
The blob	Large class, Divergent change, Data class, Lazyclass, Inappropriate intimacy
Lava flow	Code smells within the *Dispensables* taxonomy group
Functional decomposition	Code smells within *Object-orientation abusers* taxonomy group
Poltergeists	Data class and Lazy class
Golden hammer	-
Spaghetti code	Large class, Long method, Divergent change, Feature envy, Inappropriate intimacy, and Temporary field
Cut and paste programming	Duplicated code and Shotgun surgery
Continuous obsolescence	Incomplete library class
Ambiguous viewpoint	Code smells within *Object-orientation abusers* taxonomy group
Boat anchor	Speculative generality
Dead end	Incomplete library class
Input kludge	-
Continuous obsolescence	-
Continuous obsolescence	-

2.2 Practitioner's perception of refactoring drivers

This section presents the literature's view on how practitioners perceive and treat refactoring drivers, i.e code smells, anti-patterns and other design flaws. For this purpose, we have investigated relevant surveys and empirical studies, which provide developer insights on the perception and usage of these drivers.

The examined quantitative studies where selected based on their relevance to one of our research questions; *RQ1.2: What quantitative findings does the literature reveal regarding refactoring drivers?* Based on this research question, the inclusion and exclusion criteria were formed.

Inclusion criteria:

- Primarily, any field study that observes viewpoints and actions of practitioners regarding software refactoring decisions.
- Secondly, particular studies with findings on how practitioners perceive and treat refactoring drivers, i.e. code smells, anti-patterns and other quality flaws.

Exclusion criteria:

- Studies with no quantitative findings.
- Studies that do not evaluate refactoring drivers or solutions.

After defining the selection criteria, we have identified eight relevant studies. Three of which are surveys that present how practitioners perceive refactoring drivers. The remaining five are empirical studies that present evidence on to what extent these drivers are considered by practitioners. Table 12 outlines these quantitative studies. Note that the listed surveys and empirical studies are coded either with the prefix *SUR* or *EMP*.

In order to understand how practitioners perceive and treat refactoring drivers, the identified studies were examined in three aspects. Table 13, presents these aspects with corresponding studies. Now, we elaborate on these aspects.

2.2.1 Know-how and the perceived usefulness of drivers

We have examined a single survey presenting practitioner views related to how much they know about and how useful they see refactoring drivers, i.e. code smells and anti-patterns.

Table 12: Quantitative studies on software refactoring drivers

Code	Study
SUR1	*Do developers care about code smells? An exploratory survey* [23]. Researchers have conducted a survey on 85 software professionals and have investigated the developer's perception of refactoring drivers.
SUR2	*A field study of refactoring challenges and benefits* [22]. Kim et al. have conducted a survey with over three hundred professional software engineers to revealed the symptoms of code that help developers initiate refactoring.
SUR3	*Understanding the longevity of code smells: Preliminary Results of an Explanatory Survey* [20]. An explanatory survey was conducted aiming at better understanding the longevity of code smells in software projects. Researchers have asked developers to rank 5 common refactorings in terms of frequency, difficulty and importance.
EMP1	*Does the modern code inspection have value* [9]. Researchers have investigated the importance of code inspections and have identified how developers treat soft maintenance issues as refactoring drivers during code inspections to improve maintainability.
EMP2	*The evolution and impact of code smells: A case study of two open source systems* [18]. The effect of code smells on software change behaviour was investigated by analyzing the historical data of two open-source projects. Researchers have evaluated two of the Fowler's smells [6]; large class and shotgun surgery.
EMP3	*An exploratory study of the impact of code smells on software change-proneness* [17] Researchers have investigated the releations between 29 code smells and software change behaviour by statistically analyzing several releases of two open-source projects.
EMP4	*Investigating the evolution of bad smells in object-oriented code* [19]. The past versions of two open-source projects were inspected and the evolution of code smells in code were investigated. Three of Fowler's smells [6] were evaluated; long method, feature envy and switch statements.
EMP5	*Refactoring practice: How it is and how it should be supported-an eclipse case study* [16]. A case study was conducted on the structural evolution of the Eclipse project. Several code releases were inspected and the fraction of code modifications related to structural refactorings were identified.

Table 13: Study aspects and examined relevant literature

Studies	Know-how/usefulness	Criticality	Recurrence
SUR1	•	•	•
SUR2	-	•	-
SUR3	-	-	•
EMP1	-	-	•
EMP2	-	•	-
EMP3	-	•	-
EMP4	-	•	-
EMP5	-	•	-

Legend: "•" indicates relevance in each aspect

In *SUR1*, researchers have asked 85 software professionals about their know-how regarding these drivers. Only 18% of the respondents stated that they had good understanding and have previously used them. Half of the respondents had some understanding but have never used code smells and anti-patterns, whereas 32% have never heard about any of the mentioned drivers. Likewise, half of the respondents saw them moderately to extremely useful and stated that they are most useful for code inspections and in error predictions.

2.2.2 Criticality of drivers

Six of the selected studies have addressed the criticality of smell existence in code. *SUR1* seeks to answer how concerned practitioners are about the existence of code smells, whereas *SUR2* seeks to answer which quality attributes are mostly addressed and refactored by developers.

The results of *SUR1* have indicated that almost all of the survey respondents were moderately to extremely concerned about code smells. Only 6% of the respondents were not concerned at all.

In the survey *SUR2*, 20% of the respondents stated that readability is the top driver for refactoring. Only 7-10% of the respondents considered refactoring drivers such as, duplication, reuse, maintainability, legacy code, testability, and performance. Finally, dependency, logical mismatch, and debugging were least considered as drivers for refactoring.

On the other hand, the empirical studies *EMP2* to *EMP5* present findings on the criticality of certain code smells by inspecting the past versions of open-source projects.

EMP2 have evaluated two of the code smells, large class and shotgun surgery. The case study findings reveal that code smells are critical in terms of change behaviour. Large classes are exposed to bigger changes and require more maintenance. Similarly, classes infected with shotgun surgery show a higher change frequency during the development lifetime.

The results of *EMP3* is similar to *EMP2*. In this exploratory study, researchers have evaluated 29 code smells during the lifetime of software development. They have indicated that classes with smells are more prone to change. In most cases the odds for classes with smells to change is 2 to 6 times higher than for classes without smells.

Researchers of *EMP4* have observed developer behaviours related to three code smells, namely long method, feature envy, switch statements. By inspecting the past versions of two open-source projects, this study have revealed that a large portion of code smells (57.7%) remain present throughout the development lifetime. This indicates that only a few number of code smells were alarming for developers. Within those few alarming cases, long methods were refactored the most by developers, whereas code fragments infected by feature envy and switch statements were considered quite less.

EMP5 presents that about 70% of structural changes may be due to refactorings. The performed refactorings involve a variety of restructuring activities, ranging from simple element renamings and moves to substantial reorganizations of containers and inheritance hierarchies.

2.2.3 Recurrence rate of drivers

Studies *SUR1*, *SUR3* and *EMP1* have evaluated the recurrence rate of refactoring drivers including, code smells, anti-patterns and soft maintanence issues.

In *SUR1*, researchers have asked software professionals to state the most recurring code smells and anti-patterns. As a result duplicated code was by far the most mentioned smell, followed by code smells and anti-patterns related to size and complexity, namely long method, large class, and the anti-pattern accidental complexity.

The explanatory survey *SUR3* conducts an online questionnaire with 33 developer answers and seeks to answer the occurrence frequency of software anomalies found in code. Developers were encouraged to rank 5 common refactoring drivers from a list of software anomalies found in code. According to developer responses the most recurring smells are duplicated code (26), long methods (23), inadequate naming for classes, methods and variables (23), long classes with too many responsibilities (14)

and classes that deeply depend on details of other classes (10).

The empirical study *EMP1* have evaluated code inspection results and have identified the refactored soft maintenance issues by developers in order to improve software maintainability. During code inspections developers have identified and refactored four main software maintenance issues:

- 47% of the refactorings where related to software documentation. These refactorings were related to clarification, correction and documenting future work.
- Style was the second most recurring refactoring having 46%. These refactoring category consists of, code clean-up, renaming, debugging and cosmetics.
- Portability was the third most recurring refactoring type with only 5%.
- Safety related maintenance issues were only identified and refactored in the 2% of the inspections.

3 Research methodology

The main objective of this study is to gain an understanding on how refactoring decisions are made. In particular how it is described in the literature versus how it is perceived by software developers.

The objective is targeted in terms of a) studying relevant literature to form a basis on refactoring, particularly on refactoring drivers, and b) finding new empirical evidence on how developers make refactoring decisions. The research questions and sub-questions are presented in Table 14. Finding an answer to RQ1 requires a

Table 14: Research questions and sub-questions

RQ1	**What does the literature say on how refactoring decision are made?**
RQ1.1	What does the literature say about the terminology used in identifying drivers for refactoring decisions?
RQ1.2	What quantitative findings does the literature reveal regarding refactoring drivers?
RQ2	**How does developers make refactoring decisions?**
RQ2.1	What are the identified refactoring drivers from the viewpoint of software developers?
RQ2.2	How does developer behaviours in making refactoring decisions relate to the literature?

literature study in two aspects. First, we conduct a review on the terminology used by the literature to identify drivers for refactoring decisions. Second, we collect and synthesize quantitative studies that presents insight on software refactoring decision. By answering RQ1, we gain a fundamental understanding on refactoring decision, in particular its underlying drivers. Eventually, it is then possible to refer to this basis in the discussion chapter. Nevertheless, stud

Finding an answer to RQ2 through a real world investigation overlaps with Yin's (2003) definition of a case study — an empirical method aimed at investigating contemporary phenomena in their context. Similarly, the findings will act as the second building block to identify how our empirical findings relate to the literature.

The relationship between conducted studies and these research questions are presented in Table 15.

Table 15: Relation between the research questions and conducted work

Research question	Literature review	Empirical study
RQ1.1	•	-
RQ1.2	•	-
RQ2.1	-	•
RQ2.2	•	•

Legend: "•" means a study has been conducted; "-" means inapplicable

3.1 Case company

The case study was conducted in a lean start-up ICT company based in Finland. The company is producing telecommunication application services enabling video calling from multiple endpoints, including web browsers, smart TV's and mobile devices.

Software under study

The service platform consists of a server backend and multiple client endpoints. Within the scope of this study, we inspect three of the components, including the server backend, the Android client and the web client as they were actively developed during the period of this case study.

The server component was developed using Javascript and having NodeJS as the runtime environment. 16 presents the total amount of files and lines of code (LOC) in this component.

Table 16: Server backend; number of files and LOC

Language	Files	Blank lines	Comment	Code
Javascript	36	582	131	4059
Json	4	1	0	251
Python	5	72	46	213
Html	1	0	1	43
Shell	2	21	5	39
Sum	**48**	**676**	**183**	**4605**

The Android client component was developed using Java. The totam amount of files and LOC in this component is presented in Table 17.

Table 17: Android client; number of files and LOC

Language	Files	Blank lines	Comment	Code
Xml	707	2687	4921	25950
Java	143	2699	5873	15078
Shell	2	50	21	237
Sum	**852**	**5436**	**10815**	**41265**

Table 18 presents the details of the web client component. The web client was developed using the AngularJS web application framework.

Table 18: Web client; number of files and LOC

Language	Files	Blank lines	Comment	Code
Javascript	37	542	161	3631
Less	15	275	10	1359
Html	24	58	17	540
Json	2	0	0	0
Sum	**78**	**875**	**188**	**5606**

Development process

The organization is a lean startup, following lean development principles and employing most of the agile practices. Practices worth mentioning include; continuous integration and deployment, continuous improvement (process and software quality), efficient and face-to-face communication, peer reviews, one peace flow, pair programming, testing as an integral part of development, and collective code ownership.

As it is a key principle of lean development, in ensuring continuous integration and deployment, continuous software quality improvement takes place routinely but critically within the organization. In particular, software refactoring serves as the main approach in this context.

Continuous software quality improvement takes place routinely and critically within the organization. It is considered as a key principle in ensuring continuous integration

and deployment. Particularly, software refactoring serves as the main approach in this context

The empirical study was introduced to the developers during continuous process improvement meetings, namely retrospectives. As part of the case study, developers were asked to put addition attention when reporting/committing their daily progress in to the source-code revision control system RCS. Collective code ownership and peer reviews were routine practices at present. Therefore, developers were already used to report their daily commitments to the RCS.

Developers

All developers within the organization participated in the empirical study. The software development experience of the developers were varying between 3 to 10 years.

During the collection of the empirical data, 6 developers have marked their refactoring related daily commitments into the version control system.

3.2 Data collection and analysis

Data was gathered from a single source. The single source of information was stored in the version control system. Developers were asked to mark and commit their refactoring related tasks into the Git version control system. Developers were already used to this process since collective code ownership and peer reviews were routine practices.

Prior to the empirical data collection kick-off, one group discussion was conducted explaining developers the convention they are supposed to use in marking their refactoring related commits. In order to ease the data collection and analysis process, developers were informed to label relevant commits with the "refactor" keyword. This convention was used during the data collection period of 2 months.

Since Git was used as the version control system, developers marked refactoring tasks using git commit messages and git code line comments. Subsequently, a public API offered by a cloud-based git repository service was used to retrieve the commit messages, commit changes, and code line comments. The API offered Json as a response format. Therefore, it was trivial to collect, filter, and categorize refactoring tasks based on developer notes.

In the cases were developer notes lacked level of detail on the driver behind a refactoring decision, informal communication was used to resolve the ambiguity. Thanks to the use of a) peer reviews with short feedback cycles, b) efficient and face-

to-face communication, and c) technical reviews following each short development iterations, ambiguous developer notes were minimized.

The data were exported and filtered out from the version control system using a custom script developed by the researcher. This script was able to retrieve refactoring related code changes and their description from the version control system.

Once the empirical data were retrieved, they were coded for further analysis. As described by Miles and Huberman [2], data coding can be either pre-formed, e.i. codes are formed before the data is analyzed, or post-formed, e.i. codes are formed during the coding process. This study have utilized post-formed refactoring drivers as codes. However, prior to the data analysis, we have studied the literature and gathered an understanding on existing drivers. This pre-step have supported the post-formed coding process.

Having a coding process, the data analysis was performed based on the filtered data using. As described by Miles and Huberman [2], data coding can be either pre-formed, e.i., codes are formed before the data is analyzed, or post-formed, e.i., codes are formed during the coding process.

In this study, we used post-formed codes. However, prior to the data analysis, we studied the literature and gathered an understanding refactoring drivers. This pre-step supported the post-formed coding process.

The retreived data were coded with 17 post-formed refactoring drivers. In the process of the analysis, these drivers were further grouped in the four categories. Table 19 presents the refactoring driver groups and driver data codes.

Table 19: Refactoring driver groups and data codes

Driver group	**Data code**
Modifiability	Replace data clumps with objects
	Encapsulation
	Design inheritance
	Extract method or class
	Delegate lazy code
	Decoupling
	Unify code
	Remove dead code
Robustness	DevOps
	Performance optimization
	Simplify speculative complexity
	Debugging
	Algorithm change
Style	Code formatting
Documentation	Name change
	Comments
	Modify code hierarchy

4 Results and discussion

This section embodies the case study findings, accompanied by a discussion on how our findings relate to the literature.

4.1 Refactoring ratio

As discussed in the research methodology, developers were asked to label their commit messages using the keyword "refactor". As a result, we have collected data from the version control system for a period of 2 months. Table 20 presents the total amount of commits compared to the total amount of refactorings during the data collection period.

Table 20: The number of refactoring commits compared to the number of all commits

	Server	**Android**	**Web**	**Total**
All commits	156	426	539	1121
Refactoring activities	91	174	38	303
Commits with refactoring	63	132	31	226
Commits with refactoring (%)	40.40	31.00	5.80	20.20

During the data collection period, a total of 1121 changes were committed by the developers. This total commit number contains any kind of code change regarding functional or structural improvements. The findings indicate that 20% of the code change (commit) contains refactoring activities. In addition, the results imply that every individual commit may contain one to many refactoring activities.

Another interesting finding is that, even though more effort was spend on the Android client, the server code was proportionally exposed to more refactoring activities. This is due to the Android client and server having different natures in terms of programming language and development environment. The possible causes of this difference will is elaborated as we explain each identified refactoring driver.

4.2 Identified refactoring drivers and driver groups

As described in the data analysis, post-formed codes were identified during the coding process. As a result, refactoring activities were grouped under 17 post-formed refactoring drivers. We have then grouped these drivers in four categories; Documentation, style, robustness and modifiability.

Our semi-custom classification of refactoring drivers is tailored based on two sources; relevant frameworks found in the literature and the identified drivers of our empirical study.

Researchers have evaluated refactoring drivers having similar classifications. Siy and Votta [9] have divided soft maintenance issues into four categories; *documentation, portability, safety* and *style*. There is a perfect match between two studies among the driver groups *documentation* and *styles*. In respect to *safety*, our results only consists of few security related refactorings and are included in the robustness group. Likewise, our results included a limited number of refactoring drivers that could be considered related to portability or modifiability.

In another study, Mäntylä and Lassenius [15] have grouped refactoring drivers in a closely similar manner. They formed four classes; *documentation, general, structure*, and *visual representation*. Again, their *documentation* and *visual representation* categories are covered in our documentation and style groups. Significant amount of the drivers they have grouped under *structure* can be found in our modifiability category, whereas drivers related to code complexity and poor algorithms fit in our robustness category.

Our custom classification of refactoring drivers are

Figure 1 presents the number and distribution of refactorings performed per driver group.

After going through all identified refactoring drivers and classifying them meaningfully, we categorized them in four groups, namely *Documentation, Style, Robustness*, and *Modifiability*. Figure 1 presents the number and distribution of refactoring activities per each driver group.

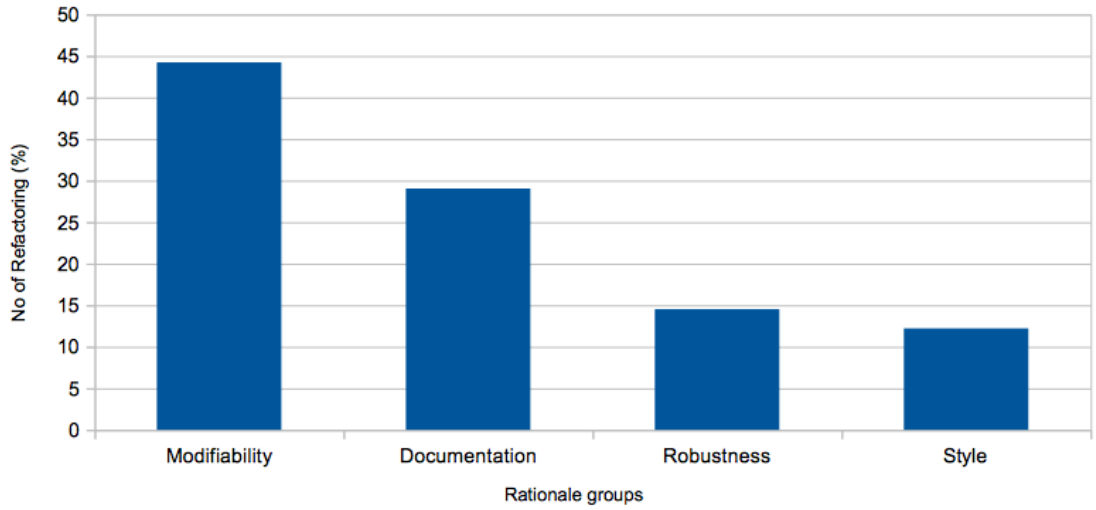

Figure 1: Distribution of the used driver groups

An overview of the driver groups, individual drivers, and the percentage of refactorings per each driver is presented in Figure 2.

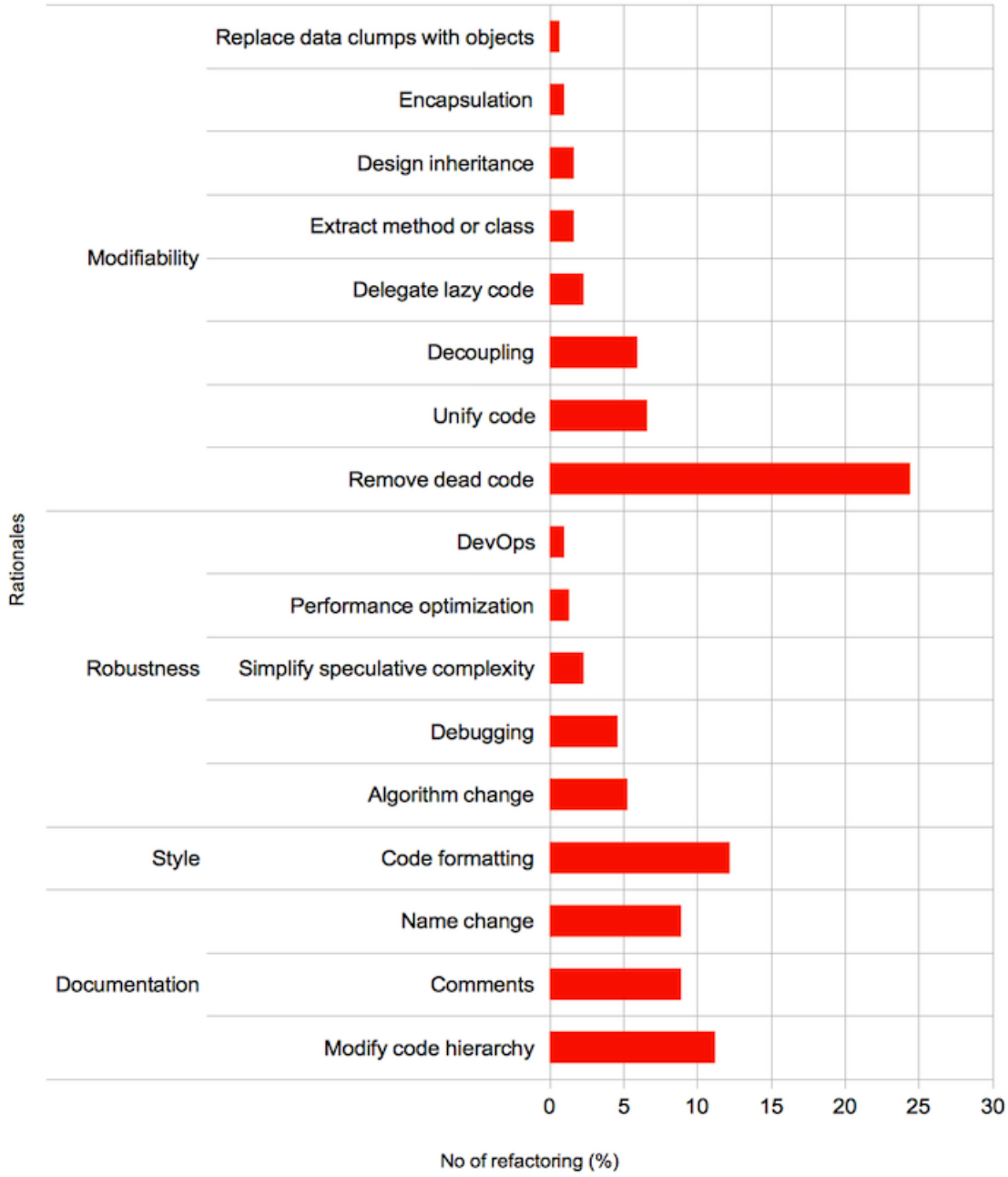

Figure 2: Distribution of the used drivers

4.2.1 Documentation

Documentation related drivers include, *Comments*, *Modify code hierarchy*, and *name change*. This group of drivers highly effect the understanability and readability of

code.

Comments

Add, remove or update line, method, class, general comments and pseudocode to improve understandability.

Together with the *Name change* driver, developers have selected *Comments* as the 4th significant refactoring activity consists of the 9% of all refactoring activities.

Modify code hierarchy

The intention of this driver is to improve class, method, conditional structure by grouping code or other minor structural changes to improve readability.

One can argue that modifying the code hierarchy can be grouped under the *Modifiability* drivers. However, modifiabilty related drivers contain structural changes in order to satisfy object-oriented design, whereas, modifying the code hierarchy is specifically concerned about improving code readability.

This driver occurs as the third most significant cause of all refactoring activities. Furthermore, this driver has been selected by the developers as being the most relevant refactoring activity in order to resolve modifiability concerns.

Name change

This driver implies changing variable, method, class, file, package naming to improve code understandability.

Together with the *Comment* driver, developers have selected *Name changes* as the 4th significant refactoring activity with 9%.

One interesting finding is the effect of tool support on how developers make name changes. Developers were able to use a powerful IDE when developing the Android client, whereas when implementing other components they were using rather elementary tools with limited capabilities. On the one hand, this had allowed developers to confidently make large amount of name changes per commit, however on the other hand, they were only able to make minor refactorings per commit. This distribution is presented in Figure 3

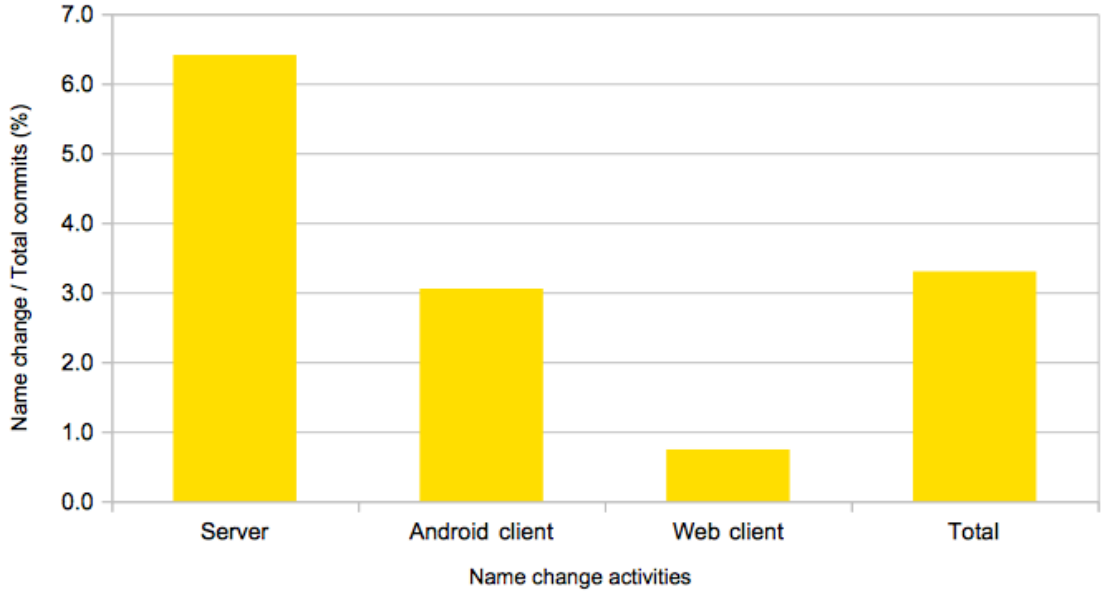

Figure 3: Name change driver distribution

4.2.2 Style

This group only contains the *Code formatting* driver. Even though this rationale is quite related to documentation rationales, they differ as styling does not require any structural changes.

Code formatting

The intention of this rationale is to improve code format – remove/add blank lines, modify indentation, modify layout - for better readability.

The second most appearing refactoring driver is *Code formatting* with 12%. Even though, code formatting and style changes has less effect on software maintainability, this driver was highly practices by the developers. This indicates that visual representation is a valid concern among developers, also knowing the fact that it is a refactoring type that is less time consuming. Another interesting finding is how *Code formatting* driver use varies between different software components. The findings indicate that while this driver was widely practiced on the server component, it was rather moderately practiced on the Android client component. Figure 4 presents this distribution.

It can be noticed that code formatting activities on the server component is considerably higher as for other components. This is due to the programming language and development environment differences between the Android client and the others. The

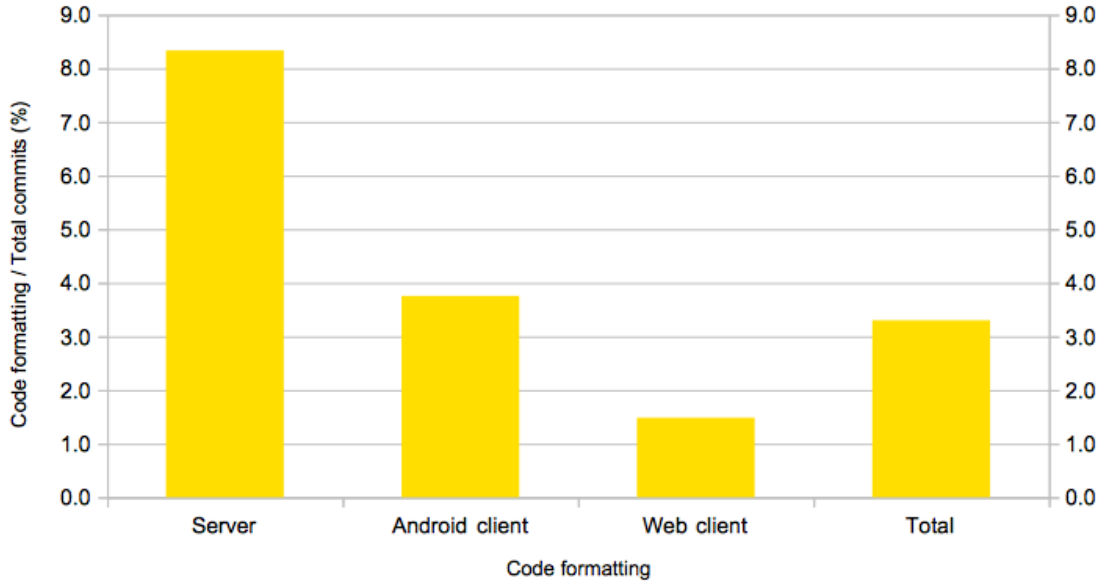

Figure 4: Code formatting driver distribution

development of the server and web component is performed using Javascript as the main programming language. Furthermore, even though somewhat specialized editors are used, as a result they do not provide comprehensive capabilities for automatic code organization. On the other hand, the Android client is implemented using Java programming language which is rather structured in nature compared to Javascript programming language. Also, available IDEs for Java and Android development, provide highly comprehensive capabilities to support code organization. Even though the web client shares the same nature as the server component, there were too less data points on this component to make healthy comparisons. In conclusion, refactoring activities related to code formatting, more generally related to the visual representation of the code, is highly related to the available tool support.

4.2.3 Robustness

The main goal of this group of drivers is to grant a solid state to the code. Robustness drivers are; *Algorithm change*, *Performance optimization*, *Simplify speculative complexity*, *Debugging*, and *DevOps*.

Algorithm change

The intention of this driver is to improve code algorithm, remove excessive nesting, simplify complexity or add/remove additional parameters/libraries to improve understandability and robustness. This driver contains 5% of all refactoring activities.

Performance optimization

The intention of this driver is to improve code algorithm and reduce computational complexity or add/remove additional parameters/libraries to improve performance. Developers used this driver in a bit higher than 1% of their refactoring activities.

Simplify speculative complexity

The intention of this driver is to simplify code based on speculative generality, misunderstood requirements and over design. This type of activities were practiced 2%.

Debugging

The goal of debugging is to add, remove, update debugging related comments and logs, or any other improvements to help development. Debugging driver contains 4,5% of refactoring activities.

DevOps

This driver consists of any code improvement or configuration to help development and operational tasks. This driver was used in 1% of the development activities.

4.2.4 Modifiability

The modifiability group of drivers has the highest occurrence. This category includes any kind of structural change which affects quality attributes such as, modularity, maintainability, readability, and generally object-orientation. Including drivers are; *Delegate lazy class*, *Decoupling*, *Extract class or method*, *Unify code*, *Encapsulation*, *Remove dead code*, *Design inheritance*, and *Replace data clumps with objects*.

Especially refactoring drivers that are motivated to resolve concerns related to object-oriented design issues where highly considered on the Android client component compared to the other component. This clearly indicates the developer's usage of object-orientation on different programming languages. Refactoring drivers in this category can be listed as follows:

- Unify code
- Replace data clumps with objects

- Extract method or class
- Encapsulation
- Design inheritance
- Delegate lazy code
- Decoupling

Figure 5 indicate that developers have considered object-orientation a higher priority on the Android client component which uses Java as the main programming language. On the other hand, developers considered object-orientation less of a concern on the server and web component which uses Javascript as the main programming language.

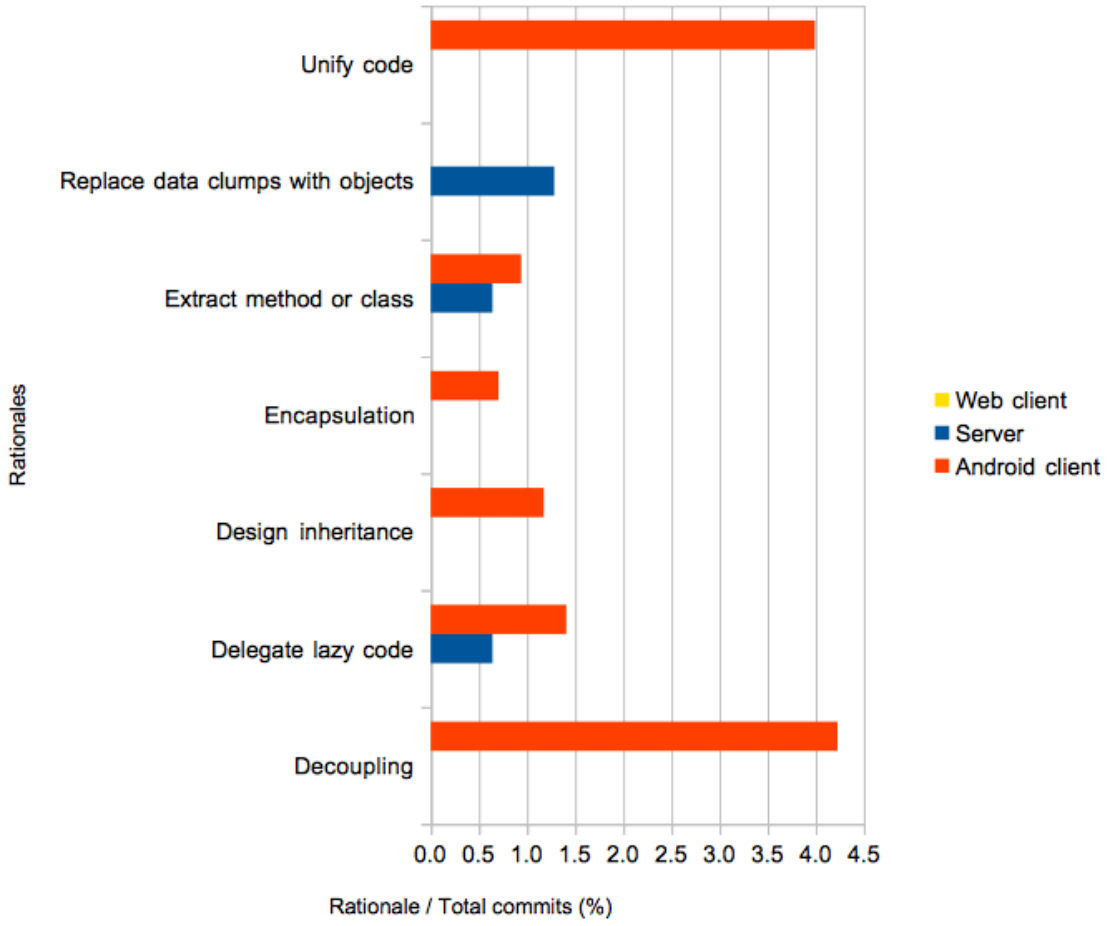

Figure 5: Object-oriented drivers and how they relate to the software components

Delegate lazy class

The aim of this driver is to delegate lazy class or method behavior to remove middle man or to remove dispensable code. This driver was considered by developers in just over 2% of the refactoring activities.

Decoupling

The idea of this driver is to decouple tightly coupled classes by applying unidirectional associations, replacing inheritance with delegation to decrease intimacy or move method to correct class to limit feature envy, eventually achieve better decoupling and improved modularity. Decoupling was used by developers in 6% of their refactoring decisions.

Design inheritance

This driver aims to improve inheritance misuse by making proper use of existing inheritance hierarchies, or using common interfaces to limit change preventers, duplication and improve maintainability. Design inheritance was considered in 1,7% of the refactoring cases.

Extract method or class

This driver aims to divide large class or long method for better readability and modularity. This driver was used close to 2%.

Unify code

The unify code driver aims to create utility function based on duplicated code, generalize existing method behavior and serve similar requests, or use constants to remove duplicated code, limit change preventers and improve maintainability. Code unification was practiced in the 6,5% of refactoring activities.

Encapsulation

Encapsulation driver redefines the scope of the class behavior and improve access rights. Encapsulation was considered in only 1% of all refactoring attempts.

Remove dead code

This driver aims to remove any unused lines remaining in code due to legacy reasons, evolving software, changing functionality and refactoring.

The distributions presented in Figure 1 and Figure 2 indicate that refactoring activities were highly focused on *Modifiability* concerns with 44% and refactoring decisions were made mostly in order to *Remove dead code* with 24%.

The context highly matters in understanding the results correctly. The empirical data was retreived from a 2 years old software. Therefore, it is understandable that refactoring activities are focusing on removing legacy code and design decisions.

Replace data clumps with objects

This driver aims to wrap related data clumps into an object, use object orientation properly to reduce data clumps, long parameter list, large classes, long methods, primitive obsession and handle code more effectively. This driver is the least occurring decision maker with having less than 1%.

4.3 Discussion

This chapter discusses the results of the study. The topics of discussion are based on the research questions. Conclusions, based on the following discussion, are presented in Chapter 5.

4.3.1 The literature's view on refactoring decisions (RQ1)

Addressing RQ1 required a literature review on the state of software refactoring, in particular on the drivers behind refactoring decisions. Therefore, it was critical to understand the existing body of knowledge on drivers for refactoring decisions. Furthermore, since the empirical part of this thesis aims to reveal developer perceptions on refactoring drivers, it was reasonable to compare our findings with related work, i.e., surveys and empirical studies.

As a result, the literature was reviewed in two portions, a) studies which present the terminology behind drivers for refactoring decisions, and b) field studies which reveal quantitative findings on refactoring, in particular its driver.

The used terminology for refactoring drivers (RQ1.1)

Our literature findings indicate that refactorings, in particular the driver behind refactoring decisions are discussed substantially in terms of code smells, which was introduced by Fowler [6] in the year 1999. However even before Fowler's book, refactoring process was practiced in the field of reengineering, in particular in software restructuring.

Following Fowler, smells in code was substantially elaborated by researchers, further analysis and classifications were developed, and quantitative studies has been conducted by means of surveys and empirical research.

One other significant contribution in the field of refactoring was introduced by Brown et al. [4]. Their book on anti-patterns was introduced even before Fowler's code smells in 1998. Brown examines anti-patterns in multiple levels including, development, architecture, and project. The development level anti-patterns introduced by Brown et al. contributes significantly to the refactoring body of knowledge.

Code smells and anti-patterns overlap and extend each other in many occasions. Development level anti-patterns reveal refactoring drivers in a higher level, which enable solutions in both code and architecture levels. Whereas, code smells help develop refactoring drivers in the code level. They especially address design problems in order to fulfill object-oriented design principles.

Code smells — help develop significant insights on where to start refactoring. However, refactoring can not be limited to what code smells indicate. Either, the definition of code smells should be extended, or other design problems should be considered together with code smells when to refactor.

Fowler describes 22 code smells and 76 refactoring techniques to overcome the described smells. Nevertheless, a significant portion of the smells only reveal object-oriented design problems. Refactoring can not be considered only when to satisfy object-orientation. There are significant amount of drivers for refactoring decisions which take e.g., documentation and code style as the main concern.

Development anti-patterns — support and cover most of the code smells. A considerable amount of code smells can be considered as symptoms of anti-patterns.

Brown et al. introduced their book *AntiPatterns: refactoring software, architectures, and projects in crisis* in 1998. Even though their book contribute significantly to the rafactoring body of knowledge, limited amount of the described anti-patterns can be considered out-dates as they discuss design problems related developers familiar with procedural programming languages. Still we can not neglect even those anti-patterns as the mentioned programming languages are still in use to a certain extent.

Fowler explains their code smells with programming examples. This is quite useful to fully understand what they really mean in each smell type, however, it is still likely that code smells are not fully understood by a conceptual definition and an example code block. Studies that classify code smells into meaningful groups [11] and analyze their usage in field studies [23], can provide an improved understanding.

On the other hand, even though they provide detailed explanations on development anti-patterns, Brown et al., [4] does not present sufficient amount of practical examples to help software professionals identify anti-patterns. The acquired understanding stays in a relatively high level compared to the acquired understanding on code smells. However, it was possible to relate anti-patterns to code smells, which aided the process of understanding drivers for refactoring decisions, the relation between code smells and anti-patterns, and their junction and disjunction points.

The quantitative studies on refactoring drivers (RQ1.2)

In seeking an answer to this research question, eight quantitative studies were examined. Three of which were surveys, while five had an empirical part.

Significant amount of the studies have also presented the risks, benefits, and effects of refactoring in general. These aspects have been considered as less relevant. Nevertheless, all identified studies were examined based on three aspects, namely the level of know-how on drivers and their perceived usefulness; the criticality of drivers; and their recurrence rate.

In terms of the level of understanding on refactoring drivers, researchers [23] state that one-third of all software practitioners attending their survey indicated that they were unaware of any refactoring driver. Similary, in terms of perceived usefulness, only half of the practitioner's found refactoring drivers useful. On the other hand, our empirical study does not include developer's statements. Rather that that, were were interested in their refactoring behaviours.

Likewise, researchers [22, 23] have presented software practitioner statements on how critical they see refactoring drivers. Again, we have no equivalent study to make a comparison. However, other studies [16–19] have presented empirical evidence. The fact that all of these studies present structural changes related to software modifiability is in line with our findings. We have identified that 45% of all refactoring decisions were made based on the modifiability driver group.

In terms of recurrence rate of refactoring drivers, the survey study of Yamashita and Moonen [23] is substantially aligned with our empirical findings. They have presented a ranking list of most considered refactoring drivers. Duplicated code is the first item in their list, followed by other code smells higly related to modifiability.

Similary, Siy and Votta [9] have classified soft maintanence issues in four groups including, documentation, style, portability and style. Issues related to documentation have occured with 47%, followed by style 46%, portability 5%, and safety 3%. Even though our driver groups are similar to Siy and Votta, the distribution is dramatically different. Our findings indicate refactoring frequency for documentation issues having 24%, style having 12%, modifiability which covers portability 44%, and finally robustness which covers safety having 15%. The dramatic difference is due to our driver groups being significantly broader than Siy and Votta's maintanence groups.

Other empirical studies found in the literature such as Olbrich et al. [18] and Khomh et al. [17] evaluate code smells and their effect to change-proneness. Since this study does not focus on change-proness, it is difficult to make comparison. Similarly, Chatzigeorgiou and Manakos [19] have studied the lifespan of code smells during an entire software development lifetime, this is again hard to relate to our findings.

4.3.2 The practitioner's view on refactoring decisions (RQ2)

In regards to the viewpoints of developers on the drivers for refactoring decisions, 17 drivers were identified. These drivers were post-formed and coded as the data analyses continued. Furthermore, the code smells [6] and their taxonomy [11] were utilized in identifying the refactoring driver codes.

In order to address RQ2, next we discuss the identified software practitioner views regarding the drivers for refactoring decisions, accompanied by a discussin on how they relate to the literature.

Practitioner's view on the drivers for refactoring decisions (RQ2.1)

Developer behaviours regarding refactoring drivers reveal results related to the effect of the used development tools and programming language.

One interesting finding was the percentage of refactorings related to code style. On the server component practitioners were using a scipting language and a relatively limited text editor for development. The findings have clearly showed how these two attributes affect the amount of refactorings. Using text editors with limited capabilities have led developers to make refactorings in small chunks. Similarly, the used scipting language did not allow them to make two-stage debugging due to the lack of a compiler. This again led the developers to make smaller changes at a time. As presented in Figure 6, style (code formatting) refactorings were the second highest drivers for refactoring decisions.

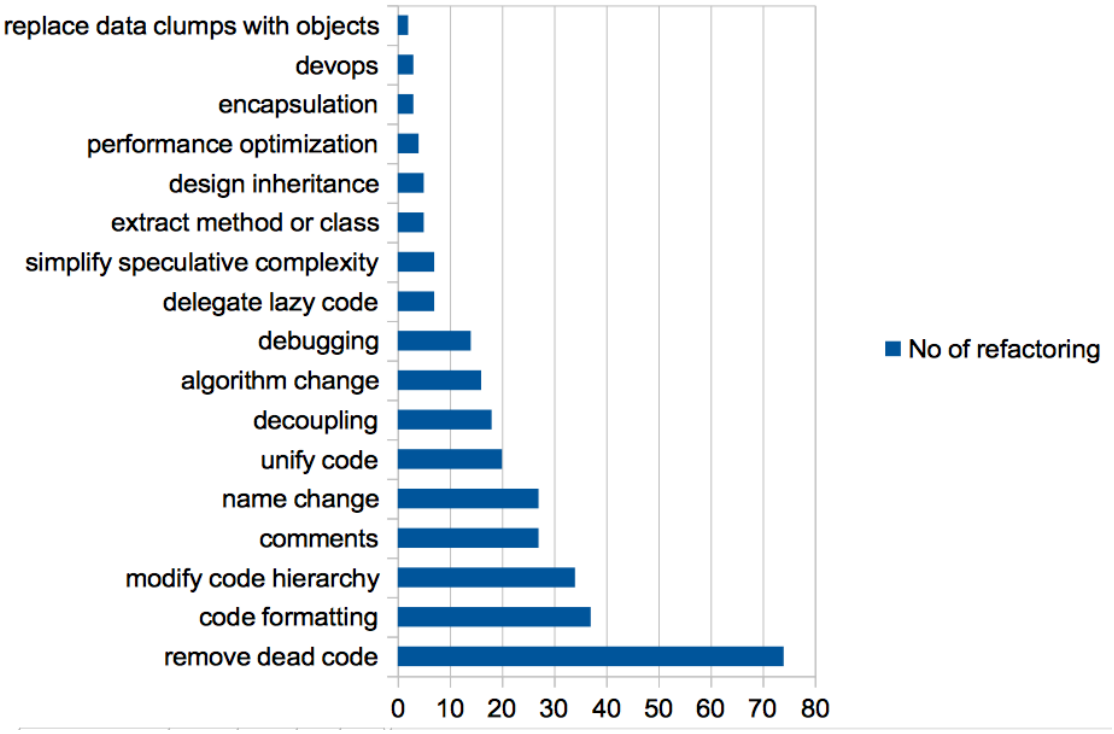

Figure 6: Number of refactorings per driver

Comparing the empirical findings with the literature (RQ2.2)

The empirical data were analyzed and then compared with Fowler's code smells [6] and with the code smell taxonomy of Mäntylä et al [11].

Table 21 presents how the identified refactoring drivers relate to the Fowler's code smells [6]. Although half of the refactoring activities were able to match with code

smells, dramatic amount of activities did not match with any smell. Significant amount of refactorings did not match any of the listed code smells.

The driver *Remove dead code* was the most recurring unmatched refactoring. Almost 25% of the refactoring decisions were made in order to remove dead code. The second largest uncategorized driver was *Code formatting*. These two drivers form more that half of the total amount. Neither Fowler's [6] code smells nor Brown's [4] anti-patterns describe these two in making refactoring decisions. Nevertheless, other studies that mention these drivers exist [9, 15].

Table 21: Identified drivers and the code smell taxonomy comparison

Code smell	**refactorings (%)**
uncategorized	55,80
comments	25,50
feature envy	4,00
shotgun surgery	3,00
duplicate code	2,60
inappropriate intimacy	2,60
temporary field	1,70
middle man	1,30
speculative generality	1,30
alternative classes with different	1,00
data class	1,00
large class	1,00
data clumps	0,70
lazy class	0,70
long method	0,70
refused bequest	0,70
primitive obsession	0,30
switch statements	0,30

Similarly 7 presents how the identified refactoring drivers relate to the code smell taxonomy of Mäntylä et al. [11]. The uncategorized category is relatively lower than the code smell comparison. This is due to the fact that we have considered *Remove dead code* driver as a *Dispensable*.

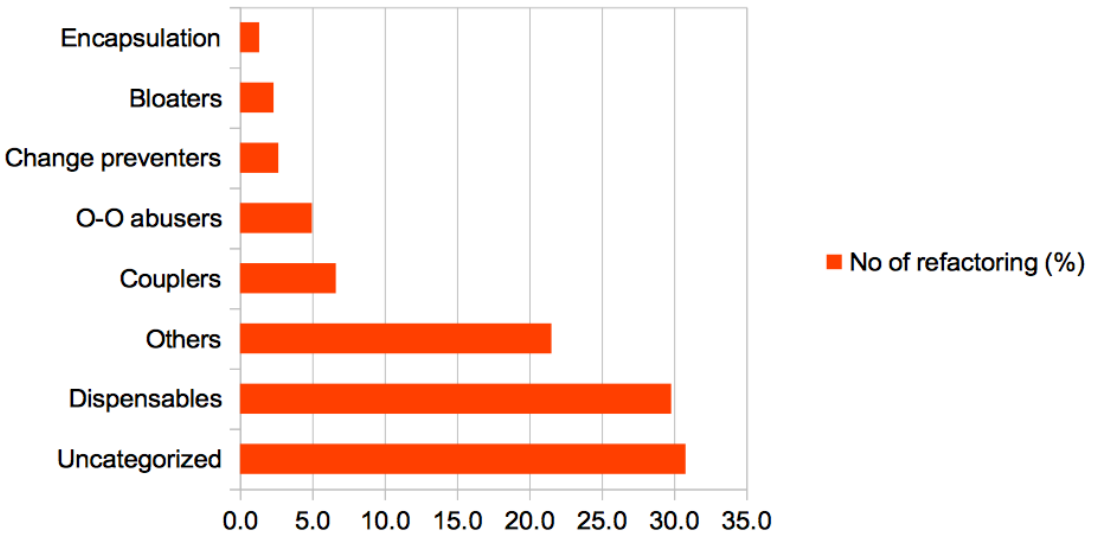

Figure 7: Number of refactorings per code smell taxonomy

5 Conclusions

This chapter presents the conclusions of the study, indicates the potential threats to validity, and suggests the most significant areas for further study.

The main purpose of this study was to gain an understanding on refactoring decisions, in particular how it is described in the literature versus how it is perceived by software practitioners.

We have studied the state of software refactoring in the literature and have investigated the software practitioner's view on refactoring decisions. This is performed in terms of a) studying relevant literature on the drivers for refactoring decision, and b) conducting a quantitative study to reveal new empirical evidence on the practitioner's perception of refactoring drivers.

The literature findings indicate that refactoring decisions can be made by using the mentioned refactoring drivers. Code smells and anti-patterns were the most referred drivers in the on making refactoring decisions, particularly in the object-oriented context.

22 code smells of Fowler [6], have described the most significant design problems which occur in object-oriented context and recommends refactoring solutions for each smell. On the other hand, Brown et al., have introduced 14 development anti-patterns and proposes refactoring solutions having a higher abstraction level.

Development anti-patterns in isolation do not offer a practical reference to determine refactoring drivers. However, the introduced anti-patterns provide significant insights on design decisions. Additionally, code smells can be considered as being symptoms of anti-patterns. Understanding the relation between anti-patterns and code smells can aid the process of making refactoring decisions.

Even though, code smells and development anti-patterns can be used as a fundamental parameter for decision making, there are only few empirical evidence on how software practitioner's make use of these drivers.

Surveys and empirical studies has been conducted in the field, whereas only few of them investigate the usage of code smells and anti-patterns. Hence, the empirical part of this thesis have aimed to study this unknown and reveal quantitative findings.

As a result of the case study, 17 refactoring drivers were identified. These drivers were then compared with the code smells found in the literature. They were not compared to anti-patterns, as anti-patterns have a higher abstraction level compared to the identified findings.

The comparison between identified refactoring drivers and code smells indicate that, Fowler's smells were used in making only half of the refactoring decisions in the case

company. Even though the identified drivers related to modifiability and robustness had a good match with Fowler's smells, drivers related to documentation and style did not match at all. Considering the total of documentation and style related refactoring activities consists of the 45% of all, it is clear that code smells are not enough to base refactoring decisions. They should be extended with additional smells to be able to provide a good refactoring coverage.

We recommend software practitioners to improve their understanding on drivers for refactoring decisions. They should be aware of the most critical design flaws and organize their refactoring decisions accordingly. The identified refactoring drivers and driver groups of this study can provide insights to software practitioners in this matter.

There are significant amount of studies investigating the possibilities of tool support for refactoring activities. One of our findings indicate that, developers make good use of development tools which automates refactoring tasks. However, tool support is still limited to documentation and styling related refactoring activities. Refactoring tasks related to modifiability and robustness are still best accomplished with manual effort.

5.1 Threads to Validity

The quality of this study can be questioned in terms of internal and external validity.

Internal validity questions the comprehensibility [2]. In our case, this can be a threat when considering the maturation of the developers during the case study period. At the case study kick-off, developers were informed to mark, elaborate and report their daily activities which they think relates to software refactoring. Due to this, developers could have been actively thinking and improving their understanding on refactoring decision making. This potential threat could clearly cause deviation on our findings.

One other internal validity threat can be the experimental bias. The researcher of this study was also part of the development team and therefore had an impact on the collected empirical data. Prior to the case study, the researcher had already gathered some level of understanding regarding the literature's view on refactoring drivers and decisions. Du to the researchers prior understandings and expectations on the subject, his refactoring reportings could include a level of deviation from the actual case.

External validity questions the extent of which the study is generalizable [2]. The applicability of our study findings to other contexts can be questioned on the grounds that this study was conducted on a single case company with a development team of 6 developers, on a certain software product, and having a certain development

environment.

5.2 Future Work

Even though it was not part of the main objective, the empirical findings indicate that there is a significant relation between certain refactoring drivers and the native of the used programming languages. Similar signs where identified on the relation between refactoring drivers and the surrounding development environment. Further studies can be conducted evaluation the impact of having a different development nature on refactoring decision making.

One interesting but challenging area would be to assess how the performed refactoring decisions effect the software quality characteristics (e.g., complexity, understandability, maintainability), and similarly, how they effect the development process in terms of productivity, cost and spent effort.

One final future study would be to investigate the impact of the performed refactoring decisions on lowering software costs and future changes.

References

[1] Meir M Lehman. *Laws of program evolution-rules and tools for programming management*. 1978.

[2] Matthew B Miles and A Michael Huberman. *Qualitative data analysis*. Sage Newbury Park,, CA, 1985.

[3] William F Opdyke. "Refactoring: A program restructuring aid in designing object-oriented application frameworks". PhD thesis. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[4] William J Brown et al. "AntiPatterns: refactoring software, architectures, and projects in crisis". In: (1998).

[5] Richard W Selby and Michael A Cusumano. "Microsoft secrets". In: *How the World's Most Powerful* (1998).

[6] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[7] Yoshio Kataoka et al. "Automated support for program refactoring using invariants". In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society. 2001, p. 736.

[8] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. "Metrics based refactoring". In: *Software Maintenance and Reengineering, 2001. Fifth European Conference on*. IEEE. 2001, pp. 30–38.

[9] Harvey Siy and Lawrence Votta. "Does the modern code inspection have value?" In: *Proceedings of the IEEE international Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society. 2001, p. 281.

[10] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-oriented reengineering patterns*. Elsevier, 2002.

[11] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. "A taxonomy and an initial empirical study of bad smells in code". In: *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE. 2003, pp. 381–384.

[12] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[13] Tom Tourwé and Tom Mens. "Identifying refactoring opportunities using logic meta programming". In: *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. IEEE. 2003, pp. 91–100.

[14] Tom Mens and Tom Tourwé. "A survey of software refactoring". In: *Software Engineering, IEEE Transactions on* 30.2 (2004), pp. 126–139.

[15] Mika V Mäntylä and Casper Lassenius. "Drivers for software refactoring decisions". In: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM. 2006, pp. 297–306.

[16] Zhenchang Xing and Eleni Stroulia. "Refactoring practice: How it is and how it should be supported-an eclipse case study". In: *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE. 2006, pp. 458–468.

[17] Foutse Khomh, Massimiliano Di Penta, and Y Gueheneuc. "An exploratory study of the impact of code smells on software change-proneness". In: *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE. 2009, pp. 75–84.

[18] Steffen Olbrich et al. "The evolution and impact of code smells: A case study of two open source systems". In: *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*. IEEE Computer Society. 2009, pp. 390–400.

[19] Alexander Chatzigeorgiou and Anastasios Manakos. "Investigating the evolution of bad smells in object-oriented code". In: *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. IEEE. 2010, pp. 106–115.

[20] Roberta Arcoverde, Alessandro Garcia, and Eduardo Figueiredo. "Understanding the longevity of code smells: preliminary results of an explanatory survey". In: *Proceedings of the 4th Workshop on Refactoring Tools*. ACM. 2011, pp. 33–36.

[21] Raed Shatnawi and Wei Li. "An empirical assessment of refactoring impact on software quality using a hierarchical quality model". In: *International Journal of Software Engineering and Its Applications* 5.4 (2011), pp. 127–149.

[22] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. "A field study of refactoring challenges and benefits". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM. 2012, p. 50.

[23] Aiko Yamashita and Leon Moonen. "Do developers care about code smells? An exploratory survey". In: *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE. 2013, pp. 242–251.

[24] Mesfin Abebe and Cheol-Jung Yoo. "Trends, Opportunities and Challenges of Software Refactoring: A Systematic Literature Review." In: *International Journal of Software Engineering & Its Applications* 8.6 (2014).